SAL introduction for AFM, Seattle 21 August 2006

SAL: Language, Model Checking, and Test Generation

Tutorial for Automated Formal Methods 2006

John Rushby
SAL was built by Leonardo de Moura

Computer Science Laboratory
SRI International
Menlo Park CA USA

SAL Origins

- SAL stands for Symbolic Analysis Laboratory
- Originally intended as an intermediate language between application notations such as Statecharts and model checkers such as Murϕ and SMV
- And also a common intermediate language among other tools such as invariant generators
  - A (too) early tool bus
- Developed in collaboration with David Dill, Tom Henzinger, and some Verimag input
- Murϕ died (actually, it's reportedly living in Utah)
- And translation to SMV was so arduous you might as well build a direct model checker
- So we did

SAL Language

- I'll use the Needham Schroeder cryptographic authentication protocol as an example to introduce the language
- Many of the type and expression constructions, and much of the syntax should be familiar to PVS users
- The big difference is we're specifying the system (and its environment) as state machines
- And its properties as LTL formulas

Example: Needham Schroeder

Message 1. $A \rightarrow B$: $A.B.\{A, N_A\}_{PK(B)}$

Message 2. $B \rightarrow A$: $B.A.\{N_A, N_B\}_{PK(A)}$

Message 3. $A \rightarrow B$: $A.B.\{N_B\}_{PK(B)}$.

Getting Started: the Network

- Want a "network" that is generic wrt. message type
- Acts like a one-place buffer
- Messages can be written (if empty),
- And read, copied, overwritten (if full)

Network

```
network{msg: TYPE;}: CONTEXT =
BEGIN
  bufferstate: TYPE = {empty, full};
  action: TYPE = {read, write, overwrite, copy};

network: MODULE =
BEGIN
  INPUT  act: action, inms: msg
  OUTPUT nstate: bufferstate, buffer: msg
INITIALIZATION
  nstate = empty;
TRANSITION
  ...
```

Network (ctd.)

```
[
 act' = write AND nstate = empty -->
  nstate' = full;  buffer' = inms';
[]
 act' = overwrite AND nstate = full -->
  buffer' = inms';
[]
 act' = read AND nstate = full -->
  nstate' = empty;
[]
 act' = copy AND nstate = full -->
  nstate' = nstate;
[]
 ELSE -->
]
```

The Participants

- Need at least two principals
- And an intruder
- These are subtypes of participants
- Useful to have an extra "error" id for initialization etc.

Participants

```
needhamschroeder: CONTEXT =
BEGIN
  ids: TYPE = {a, b, e, X};

  participants: TYPE = {x: ids | x /= X};
  intruder(x: participants): BOOLEAN = x=e;

  intruders: TYPE = {x: participants | intruder(x)};
  principals: TYPE = {x: participants | NOT intruder(x)};
```

Nonces

- In practice, need to make sure these are fresh
- In modeling, they can be deterministic
  - Do not endow the intruder with guessing ability

```
nonces: TYPE = ids;
nonce(a: participants): nonces = a;
```

Messages

- Messages contain an encrypted component
- When decrypted it's a triple of type dmsg
- arb is used for the initial value

```
dmsg: TYPE = [ids, nonces, nonces];
arb: dmsg = (X,X,X);
```

  Otherwise the model checker might use a "magic" value
- An encrypted message records the key used

```
emsg: TYPE = DATATYPE
  enc(key: ids, payload: dmsg)
END;
```

- An encrypted message on the network indicates its source and destination

```
msg: TYPE = [# src: participants, dest: participants,
                em: emsg #];
```

- Tuple, datatype, record, just for variety

Decryption

Can successfully decrypt an encrypted message only if you are the participant whose key was used

```
dec(k: participants, m:emsg): dmsg =
 IF key(m)=k THEN payload(m) ELSE arb ENDIF;
```

Otherwise, get arb

State of the Principals

- Initially sleeping
- May decide to initiate a dialog and go to waiting
- And then to engaged if the protocol completes

```
states: TYPE = {sleeping, waiting, engaged,
                tentative, responding};
```

- If another initiates the dialog, go to tentative
- And then to responding if the protocol completes
- In either case, responder is the identity of the other

Principals

Initially, each principal is sleeping, and its responder is set to itself

```
principal[i: principals]: MODULE =
BEGIN
  INPUT nstate: net!bufferstate, imsg: msg
  GLOBAL act: net!action, omsg: msg
  LOCAL pc: states, responder: participants
INITIALIZATION
  pc = sleeping;
  responder = i;
```

Principals (ctd. 1)

Waking up and initiating a dialog with j

```
TRANSITION
[
([] (j: participants): i /= j AND
  pc = sleeping AND nstate = net!empty -->
    pc' = waiting;
    responder' = j;
    omsg' = (# src := i, dest := j,
               em := enc(j, (i, nonce(i), X)) #);
    act' = net!write;
)
```

Principals (ctd. 2)

Waking up and responding to a dialog initiated by j

```
[]
([] (j: participants): i /= j AND
     pc = sleeping AND nstate = net!full
     AND imsg.src = j AND imsg.dest = i
     AND dec(i, imsg.em).1=j  -->
   responder' = j;
   pc' = tentative;
   act' = net!overwrite;
   omsg' = (# src := i, dest := j,
               em := enc(j,(X, dec(i,imsg.em).2, nonce(i)))#);
)
```

Principals (ctd. 3)

Initiator accepts the response from j

```
[]
  pc = waiting AND nstate = net!full
     AND imsg.src = responder AND imsg.dest = i
     AND dec(i,imsg.em).2 = nonce(i) -->
   pc' = engaged;
   act' = net!overwrite;
   omsg' = (# src := i, dest := responder,
              em := enc(responder, (X, dec(i,imsg.em).3, X))#);
```

Principals (ctd. 4)

Responder accepts second message from initiator j

```
[]
  pc = tentative AND nstate = net!full
     AND imsg.src = responder AND imsg.dest = i
     AND dec(i,imsg.em).3 = nonce(i) -->
   pc' = responding;
   act' = net!read;
[]
 ELSE -->
]
END;
```

Otherwise do nothing

Intruders

Need to provide the intruder with memory for messages it has seen but not been able to decrypt, and for the contents of messages (i.e., nonces) that it has decrypted

```
intruder[x:intruders]: MODULE =
BEGIN
  GLOBAL act: net!action, omsg: msg
  INPUT nstate: net!bufferstate, imsg: msg
  LOCAL  nmem, n1, n2: nonces, mmem: emsg
INITIALIZATION
  nmem = nonce(e);
  mmem = enc(X,(X,X,X));
```

We provide memory for one of each: nmem and mmem; n1 and n2 are temporaries

Intruders (ctd. 1)

Intruder can read and decrypt messages sent to itself

```
TRANSITION
[
 nstate = net!full AND imsg.dest = x  -->
   nmem' IN {dec(x,imsg.em).2, nmem};
   act' IN {net!read, net!copy};
```

Nondeterministically replaces saved nonce with the new one, and removes the message or copies it

Intruders (ctd. 2)

Can save whole messages not addressed to itself

```
[]
 nstate = net!full AND imsg.dest /= x -->
   mmem' IN {imsg.em, mmem};
   act' IN {net!read, net!copy};
```

Intruders (ctd. 3)

Can send remembered message to j, while masquerading as i

```
[]
([] (i: participants, j: principals): TRUE -->
   act' = IF nstate = net!empty
            THEN net!write
          ELSE net!overwrite ENDIF;
   omsg' = (# src := i, dest := j, em := mmem #);
)
```

Intruders (ctd. 4)

And can manufacture messages containing its own nonce or a remembered one

```
[]
([] (i: participants, j: principals): TRUE -->
   act' = IF nstate = net!empty
             THEN net!write
            ELSE net!overwrite ENDIF;
   n1' IN {nmem, nonce(x)};     n2' IN {nmem, nonce(x)};
   omsg' = (# src := i, dest := j, em := enc(j, (i, n1', n2'))#);
)
[]
  ELSE -->
]
END;
```

And that's all it can do

The Complete System

- Asynchronously compose some collection of principals and intruders
- And synchronously compose that compound with the network
- We'll have two principals a and b, and single intruder e
  No explicit limit on interleaved runs

```
system: MODULE =
 (([] (id: principals): principal[id]) [] intruder[e])
   || (RENAME buffer TO imsg, inms TO omsg IN net!network);
```

- We rename the buffer and inms of the network so that they connect up to the imsg and omsg of the principals and intruder

Authentication Property

- The property we wish to examine is correct authentication
- Whenever a principal x reaches the responding state with a principal y, must be that y initiated the protocol with x
  - That is, y must be in the waiting or engaged states and have x as its responder
- We specify this as the property prop

```
prop: THEOREM system |- G((FORALL (x,y: principals):
   (pc[x]=responding AND responder[x]=y) =>
        ((pc[y]=waiting OR pc[y]=engaged)
         AND responder[y]=x)));
```

Symbolic Model Checking

- Compile the model to a Boolean transition relation T
  - i.e., a circuit
- Initialize the Boolean representation of the stateset S to the initial states I
- Repeatedly apply T to S until a fixpoint
  - $S' = S \cup \{t \mid \exists s \in S : T(s,t)\}$
  - Final S is a formula representing all the reachable states
- Check the property against final S
- Mechanized efficiently using BDDs
  - Reduced ordered Binary Decision Diagrams

  Commodity software, honed by competition (CUDD)

Bounded Model Checking (BMC)

- Given system specified by initiality predicate I and transition relation T on states S
- Is there a counterexample to property P in k steps or less?
- Find assignment to states $s_0, \ldots, s_k$ satisfying

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge \neg(P(s_1) \wedge \cdots \wedge P(s_k))$$

- Given a Boolean encoding of I, T, and P (i.e., circuit), this is a propositional satisfiability (SAT) problem
- SAT is the quintessential NP-Complete problem
- But current SAT solvers are amazingly fast
- Commodity software, honed by competition (MiniSAT)
- BMC uses same representation as SMC, different backend
- If I, T and P use decidable but unbounded types, then it's an SMT problem: infinite bounded model checking

k-Induction

- BMC extends from refutation to verification via k-induction
- Ordinary inductive invariance (for P):

  Basis: $I(s_0) \supset P(s_0)$

  Step: $P(r_0) \wedge T(r_0, r_1) \supset P(r_1)$
- Extend to induction of depth k:

  Basis: No counterexample of length k or less

  Step: $P(r_0) \wedge T(r_0, r_1) \wedge P(r_1) \wedge \cdots \wedge P(r_{k-1}) \wedge T(r_{k-1}, r_k) \supset P(r_k)$

  These are close relatives of the BMC formulas
- Induction for $k = 2, 3, 4 \ldots$ may succeed where $k = 1$ does not
- Note that counterexamples help debug invariant

Model Checking Needham Schroeader

- Symbolic model checking
  - sal-smc -v 3 needhamschroeder prop

  Builds a transition relation on 150 state bits, 339,917,146 reachable states, and reports a counterexample ten steps long in about 10 secs
- Bounded model checking
  - sal-bmc -v 3 -d 10 needhamschroeder prop

  Builds a SAT problem with 40,756 nodes and reports the counterexample in under 10 secs
- Witness model checking
  - sal-wmc -v 3 needhamschroeder prop

  Also reports the counterexample, in about 40 secs
- Deadlock checking (may be unsound otherwise)
  - sal-deadlock-checker -v 3 needhamschroeder system

The Counterexample

```
Step 0: Initialization
-----------------------------------------------------------------------
Step 1: a sends message 1 to e
  pc[a] = waiting;  pc[b] = sleeping;  responder[a] = e;
  omsg.src = a;  omsg.dest = e;  omsg.em = enc(e, (a, a, X));
-----------------------------------------------------------------------
Step 2: e remembers a's nonce:  nmem = a;
-----------------------------------------------------------------------
Step 3: e (masquerading as a) send message 1 to b
  omsg.src = a;  omsg.dest = b;  omsg.em = enc(b, (a, a, a));
-----------------------------------------------------------------------
Step 4: b sends message 2 to a, but it is intercepted by e
  pc[a] = waiting;  pc[b] = tentative;  responder[b] = a;
  omsg.src = b;  omsg.dest = a;  omsg.em = enc(a, (X, a, b));
-----------------------------------------------------------------------
Step 5: e remembers encrypted part of b's message
  mmem = enc(a, (X, a, b));
-----------------------------------------------------------------------
Step 6: e sends message 2 (using remembered part) to a
  omsg.src = e;  omsg.dest = a;  omsg.em = enc(a, (X, a, b));
-----------------------------------------------------------------------
Step 7: a sends message 3 to e
  pc[a] = engaged;  pc[b] = tentative;
  omsg.src = a;  omsg.dest = e;  omsg.em = enc(e, (X, b, X));
-----------------------------------------------------------------------
Step 8: e remembers b's nonce from a's message 3:  nmem = b;
-----------------------------------------------------------------------
Step 9: e (masquerading as a) sends message 3 to b (with remembered nonce)
  omsg.src = a;  omsg.dest = b;  omsg.em = enc(b, (a, e, b));
-----------------------------------------------------------------------
Step 10: b falsely believes it has authenticated a:  pc[b] = responding;
```

The Counterexample (ctd.)

Is essentially the classic one

Message 1a.	$A \rightarrow I$:	$A.I.\{A, N_A\}_{PK(I)}$
Message 1b.	$I_A \rightarrow B$:	$A.B.\{A, N_A\}_{PK(B)}$
Message 2b.	$B \rightarrow I_A$:	$B.A.\{N_A, N_B\}_{PK(A)}$
Message 2a.	$I \rightarrow A$:	$I.A.\{N_A, N_B\}_{PK(A)}$
Message 3a.	$A \rightarrow I$:	$A.I.\{N_B\}_{PK(I)}$
Message 3b.	$I_A \rightarrow B$:	$A.B.\{N_B\}_{PK(B)}$.

Here, I_A indicates I masquerading as A, and the suffices a, and b on the message numbers indicate which run of the protocol they belong to

Repairing The Protocol

The protocol is easily fixed by including the identity of the responder in the encrypted portion of the second message (this prevents the replay of the encrypted portion of 2b in 2a)

Message $2'$. $\quad B \rightarrow A$: $\quad B.A.\{B, N_A, N_B\}_{PK(A)}$

Repairing The Protocol (ctd.)

- In SAL, we need to change the final assignment on slide 16 to the following (the X is changed to i)

```
omsg' = (# src := i, dest := j,
           em := enc(j, (i, dec(i,imsg.em).2, nonce(i)))#);
```

- The guard on slide 17 must then be changed (by addition of the third line below) to check that the message really does come from the expected responder

```
pc = waiting AND nstate = net!full
   AND imsg.src = responder AND imsg.dest = i
   AND dec(i,imsg.em).1 = responder
   AND dec(i,imsg.em).2 = nonce(i) -->
```

Model Checking Again

- Symbolic model checking
  - sal-smc -v 3 needhamschroeder prop

  This time there are 339,954,654 reachable states, and the property is "verified" in 12 seconds
  - Verification is relative to the intruder model and dimensions used
- Bounded model checking
  - sal-bmc -v 3 -d 10 needhamschroeder prop

  Finds no counterexamples to depth 10 in 25 seconds
- Witness model checking
  - sal-wmc -v 3 needhamschroeder prop

  Verifies the property in 30 seconds, and (internally) constructs a witness

Exploration

- Finding bugs and verifying are extreme examples of exploration
- In general, want to see runs that take us to interesting states or through interesting scenarios as a way of increasing our understanding and confidence in the operation of the system
- Can do this in a simulator, but have to think of all the inputs and interactions ourselves
- Supposing we had a simulator built on a model checker
- Then we could tell the model checker to find a path to an interesting state, then take over and explore in detail, and so on
- The SAL simulator does this

Exploration with the SAL Simulator

- Suppose we are interested in the scenario where the intruder spoofs both principals into thinking they are responding to the other
- We start the simulator and tell it to take us to a state where both principals are in tentative state

```
tulip:sal> sal-sim
 SAL Simulator (Version 2.2). Copyright (c) 2003, 2004 SRI
sal > (import! "needhamschroeder")
sal > (start-simulation! "system")
sal > (run! "pc[a]=tentative) AND pc[b]=tentative")
#t
```

- The #t means it succeeded

Exploration with the SAL Simulator (ctd. 1)

- Now we would like to see the intruder continue and bring both principals to the responding state

```
sal >(run!"pc[a]=responding) AND pc[b]=responding")
#f
```

- The #f means it failed
- Perplexed, we see if it can bring either to completion

```
sal >(run!"pc[a]=responding) OR pc[b]=responding")
#f
```

- Hmm! Let's restart and find a path where a is responding to the intruder

```
sal > (start-simulation! "system")
sal > (run! "pc[a]=responding AND responder[a]=e")
#f
```

Exploration with the SAL Simulator (ctd. 1)

- So let's get to a state where a is in the tentative state (which we already know is possible)

```
sal > (run! "pc[a]=tentative AND responder[a]=e")
#t
```

- Now we know that the intruder should be able to construct the message to take a to the responding state provided it knows a's nonce (which is also a)

- So let's see if the intruder does know this nonce in the current state

```
sal > (filter-curr-states! "nmem = a")
sal > (display-curr-states)
#t
```

Evidently not—the #t means the filtered set is empty (we also could have just looked at the current state)

Exploration with the SAL Simulator (ctd. 2)

- So now let's get back to where we were

```
sal > (backtrack!)
sal > (run! "pc[a]=tentative AND responder[a]=e")
#t
```

- And look for a state where the intruder knows the nonce

```
sal > (run! "nmem = a")
#f
```

- Hmm! Let's go back to the beginning and look for any state where it knows this nonce

```
sal > (start-simulation! "system")
sal > (run! "nmem = a")
#t
sal > (display-curr-states)
```

Exploration with the SAL Simulator (ctd. 3)

- It turns out the simulator has found a run where a initiated the dialog
- It seems that the intruder can learn a's nonce when a is the initiator, but not when it is the responder
- The difference between these cases is that a's nonce is in the second position of the emsg tuple in the former case, and the third in the latter
- Sure enough, the command in the relevant step of the intruder is the following

```
nmem' IN {dec(x,imsg.em).2, nmem}
```

  It should, of course, be changed as follows

```
nmem' IN {dec(x,imsg.em).2, dec(x,imsg.em).3, nmem}
```

Exploration with the SAL Simulator (ctd. 3)

- After making this change, we exit the simulator and restart it, and again check the properties of interest

```
sal > (import! "needhamschroeder")
sal > (start-simulation! "system")
sal > (run! "pc[a]=responding AND responder[a]=e")
#t
sal > (run! "pc[a]=responding AND pc[b]=responding")
#t
```

  This time, the simulator is able to find suitable paths

- We also check that the authentication property is still true using sal-smc

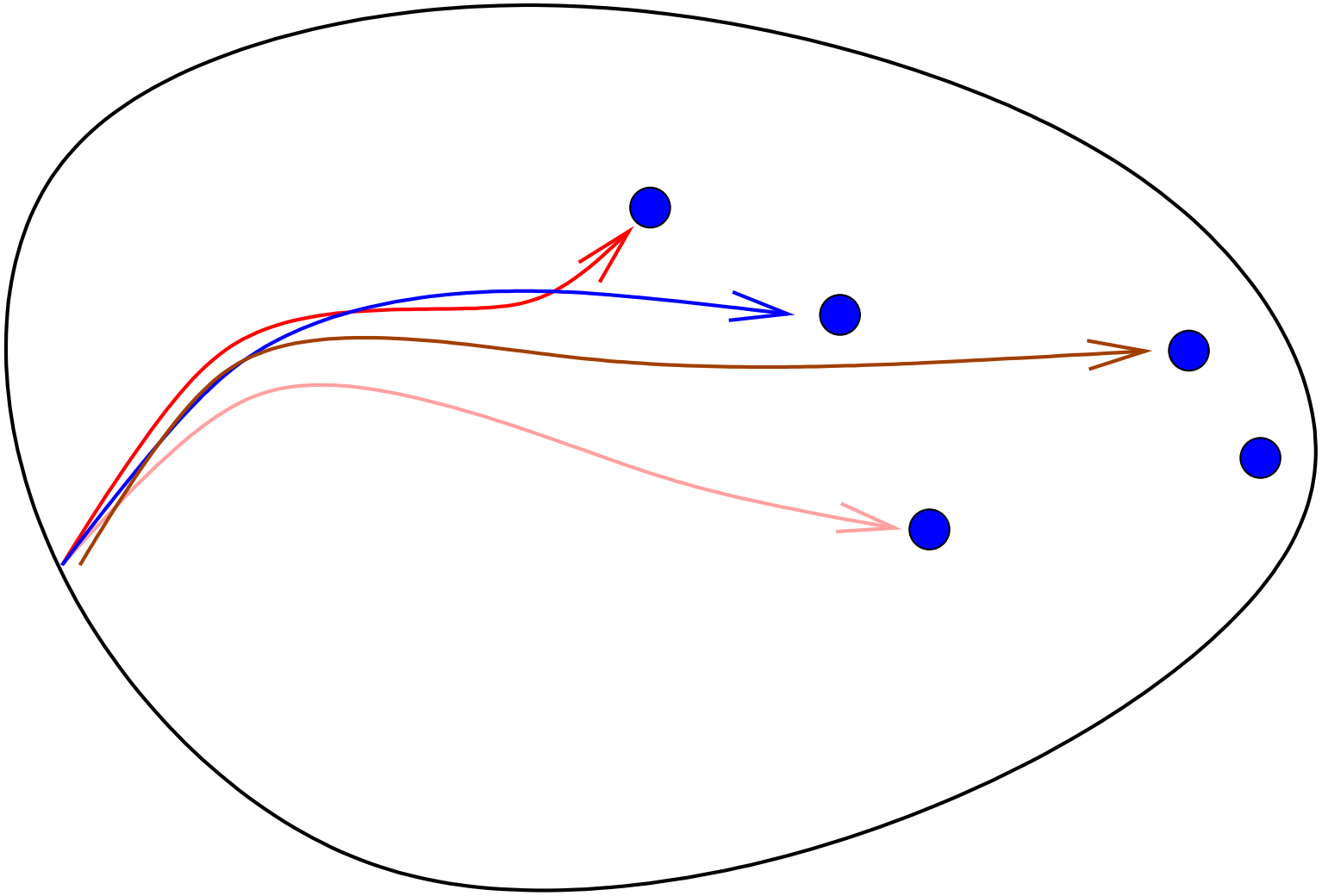
Test Generation

- Observe that counterexample to: "control cannot reach this point" is a structural test case
- So BMC can be used for automated test generation
- Actually, a customized combination of SMC and BMC works best
  - Use SMC to reach first control point, then use BMC to extend to further control points
  - Get long tests that probe deep into the system
  - Can add test purposes that constrain the kinds of tests generated
    - ⋆ e.g., Change the gear input by 1 at every step
  - Easily built because checkers are scriptable (in Scheme)

Generating Tests Using a Model Checker

- Add trap variables go TRUE when a test goal is satisfied
  - Trap variables can be inserted automatically during translation from the MBD language to the model checker
- Model check for "always not mytrap"
- Counterexample will be desired test case
- Trap variables add negligible overhead ('cos no interactions)
- For finite cases (e.g., numerical variables range over bounded integers) any standard model checker will do
  - Although many pragmatic issues concerning symbolic vs. bounded vs. explicit vs. . . for this application
  - Otherwise need infinite bounded model checker as in SAL

Tests Generated Using a Model Checker



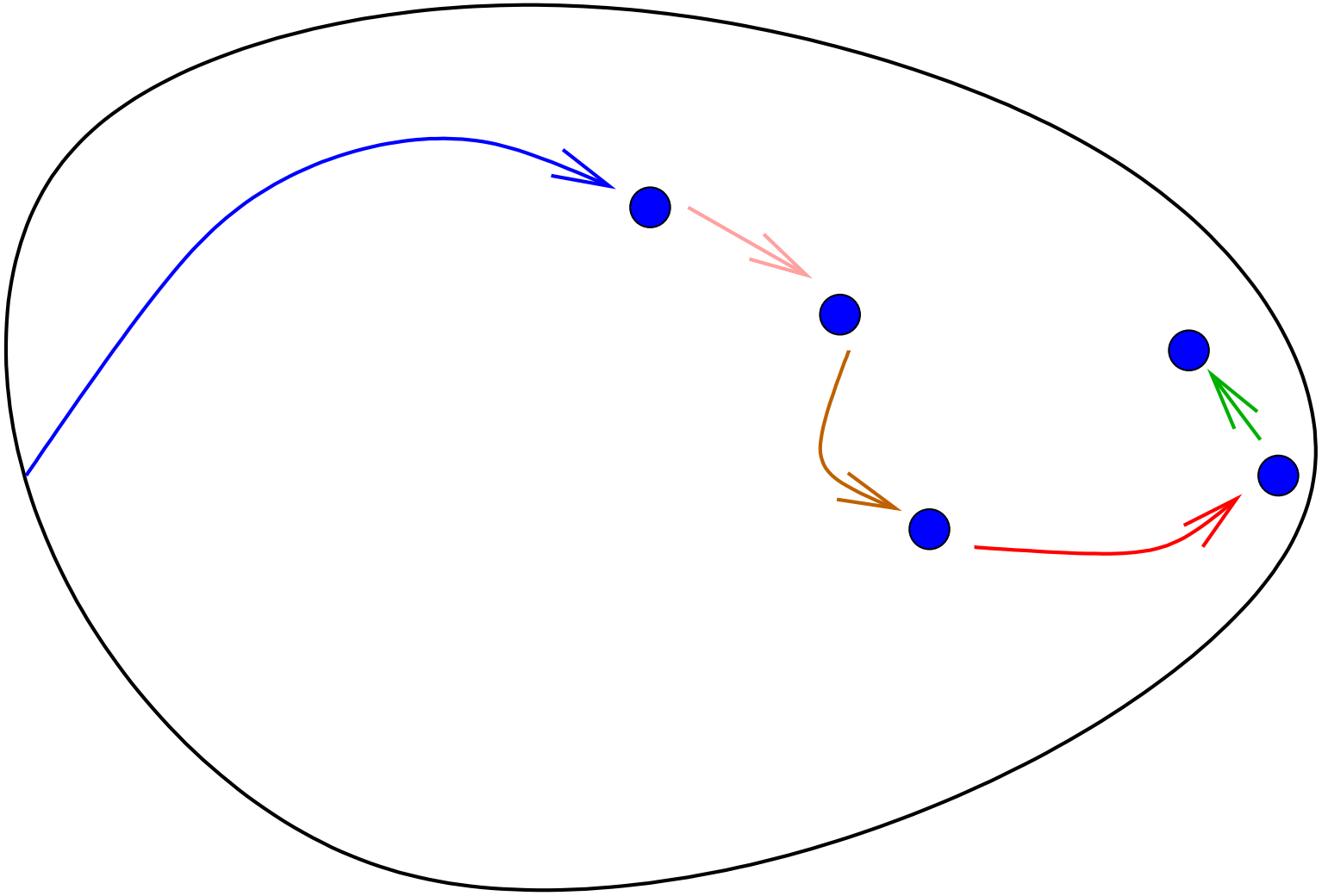
Problems Using OTS Model Checker as Test Generator

- Each test goal is treated separately: model checker is called repeatedly and performs much redundant work
- Test set has many short tests
  - Each incurs a startup cost during execution
  - Total length is large, so high execution cost
  - Much redundancy among the tests (wasteful)
  - Few long tests (so deep bugs undetected)
- Model checker may be unable to reach deep test goals

A Better Way

- Instead of starting each test from the the start state, we try to extend the test found so far
- Extending tests allows a bounded model checker to reach deep states at low cost
  - 5 searches to depth 4 much easier than 1 to depth 20
- Could get stuck if we tackle the goals in a bad order
- So, simply try to reach any outstanding goal and let the model checker find a good order
  - Can slice the model after each goal is discharged
  - A virtuous circle: the model will get smaller as the remaining goals get harder
- Go back to the start (or another earlier state) when unable to extend current test

An Efficient Test Set



Less redundancy, and longer tests tend to find more bugs

The SAL Automated Test Generator: sal-atg

- SAL is scriptable in Scheme
- `sal-atg` implements the method described in a few hundred lines of Scheme
  - (Re)starts use either symbolic or bounded model checking
    - ⋆ Parameterized choice and search depth
  - Extensions use bounded model checking
    - ⋆ Parameterized incremental search depth
  - Optional slicing after each extension or each restart
  - Customizable output to drive test harness

Core Of The SAL-ATG Test Generation Script

```
(define (extend-search module goal-list
         path scan prune innerslice start step stop)
   (let ((new-goal-list (if prune (goal-reduce scan goal-list path)
                            (minimal-goal-reduce scan goal-list path))))
    (cond ((null? new-goal-list) (cons '() path))
          ((> start stop) (cons new-goal-list path))
          (else
           (let* ((goal (list->goal new-goal-list module))
                  (mod (if innerslice
                           (sal-module/slice-for module goal) module))
                  (new-path
                   (let loop ((depth start))
                        (cond ((> depth stop) '())
                              ((sal-bmc/extend-path
                                  path mod goal depth 'ics))
                              (else (loop (+ depth step))))))))
             (if (pair? new-path)
                 (extend-search mod new-goal-list new-path scan
                                prune innerslice start step stop)
                 (cons new-goal-list path)))))))
```

Outer Loop Of The SAL-ATG Test Generation Script

```
(define (iterative-search module goal-list
          scan prune slice innerslice bmcinit start step stop)
  (let* ((goal (list->goal goal-list module))
         (mod (if slice (sal-module/slice-for module goal) module))
         (path (if bmcinit
                   (sal-bmc/find-path-from-initial-state
                       mod goal bmcinit 'ics)
                  (sal-smc/find-path-from-initial-state mod goal))))
     (if path
         (extend-search mod goal-list path scan prune
             innerslice start step stop)
         #f)))
```

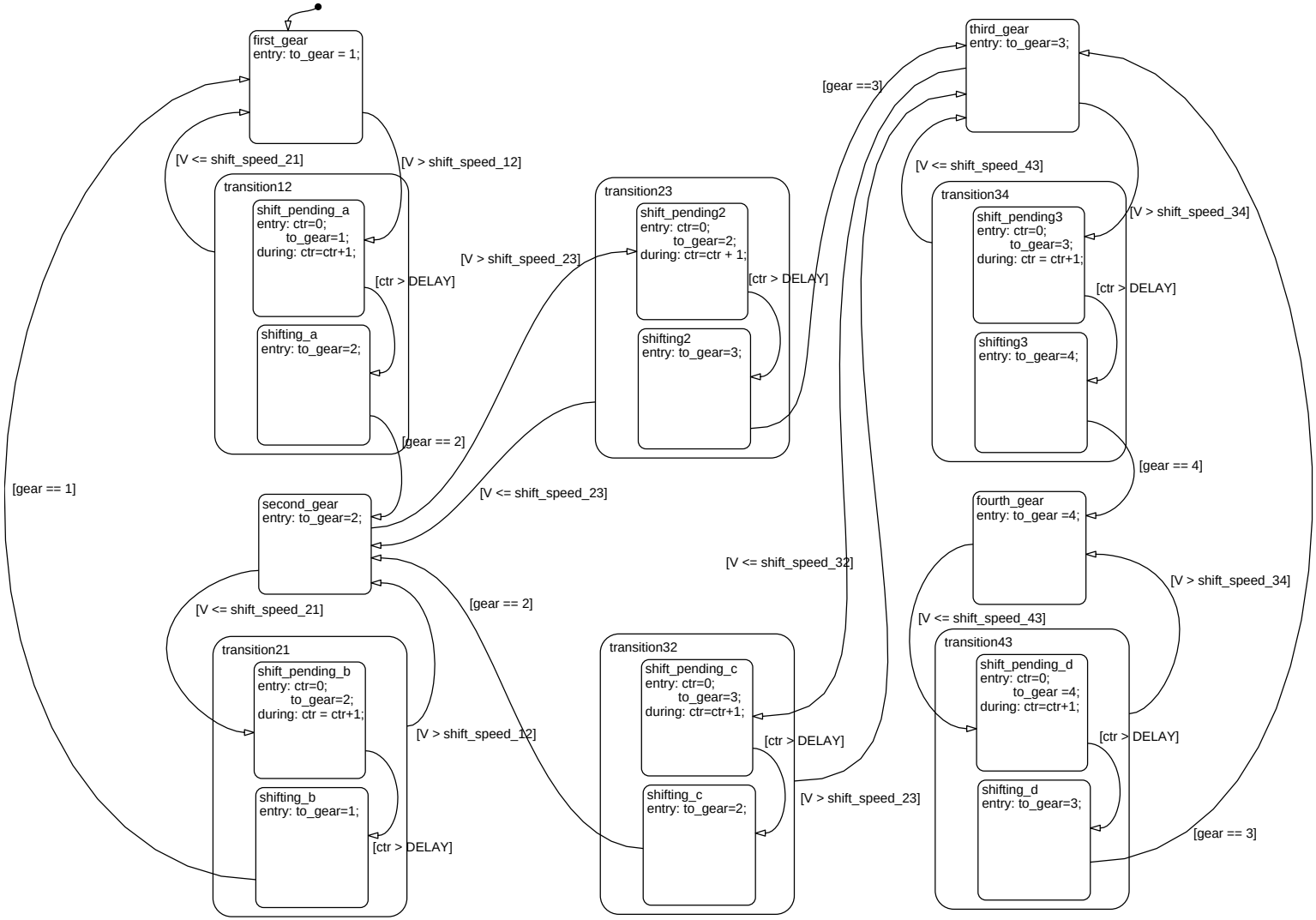
Experimental Results

- Rockwell Collins has developed a series of flight guidance system (FGS) examples for NASA
- SAL translation of largest of these kindly provided by UMN
- Model has 490 variables (576 state bits), 196 reachable control states, and 313 transitions
  - Takes 61 seconds to generate single test case of length 45 that covers all states
  - Takes 98 seconds to generate a single test of length 55 that covers all transitions
- Without extensions, get 73 tests to cover transitions: 1 of length 3, 9 of length 2, and the rest of length 1
  - Poor mutant detection
- We are in the process of testing our tests

Test Engineering with Automation

- Generating tests just to achieve structural coverage is a poor strategy
- Traditional test engineers develop tests to explore interesting cases, requirements, fault hypotheses
- We need to give them a way to do this using automation
- Specify the desired tests rather than constructing them
- Develop an observer module that sets a variable TRUE when a test has achieved some purpose
- Tell `sal-atg` to search for conjunction of each trap variable with the purpose
- In general, `sal-atg` can search for arbitrary conjunctions
  - E.g., product of structural coverage on control states and boundary coverage on some data structure

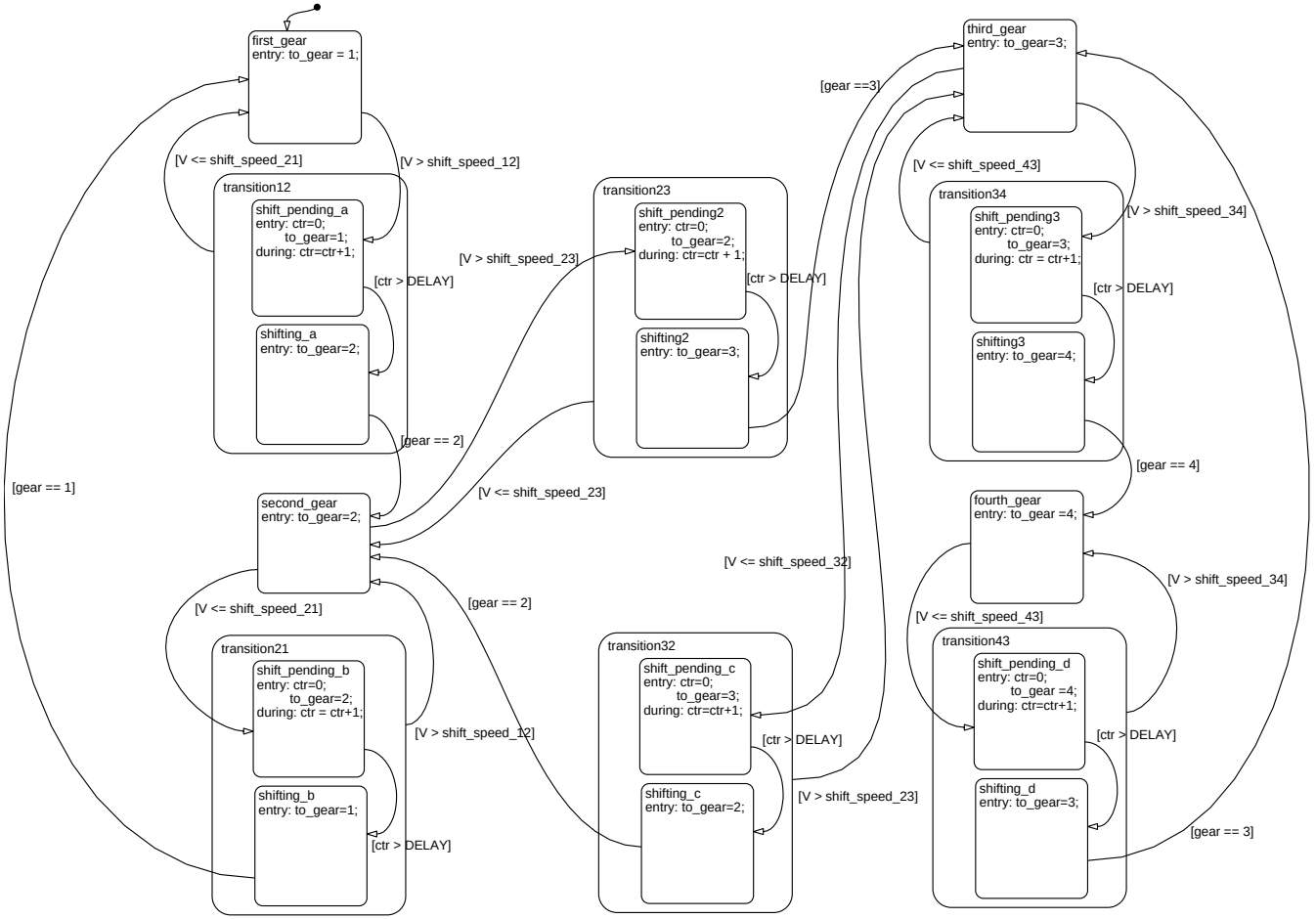
Example Shift Scheduler



Shift Scheduler

- One input is the gear currently selected by the gearbox
- Tests often change this discontinuously (e.g., 1, 3, 4, 2)
- Can easily establish the test purpose to change only in single steps, and to change at every step

Example: Shift Scheduler in StateFlow



Demo: `sal-atg -v 3 trans_ga monitored_system trans_ga_goals.scm -id 15 -ed 7 --testpurpose`